MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject Name : PYTHON PROGRAMMING

CLASS : I-MCA

SUBJECT CODE : 23PCA13

UNIT-II

Strings and Text Files: Accessing Characters and substrings in strings - Data encryption- Strings and Number systems- String methods Text - Lists and Dictionaries: Lists Dictionaries Design with Functions: A Quick review - Problem Solving with top-Down Design - Design with recursive Functions – Managing a program's name space- Higher-Order Functions.

Strings and Text Files

Word processing and program editing are obvious examples, but text also forms the basis of e-mail, Web pages, and text messaging. In this chapter, we explore strings and text files, which are useful data structures for organizing and processing text.

Accessing Characters and Substrings in Strings In we used strings for input and output. We also combined strings via concatenation to form new strings.

The Structure of Strings Unlike an integer, which cannot be decomposed into more primitive parts, a string is a data structure.

A data structure is a compound unit that consists of several other pieces of data. A string is a sequence of zero or more characters.

Recall that you can mention a Python string using either single quote marks or double quote marks. Here are some examples:

>>> "Hi there!" 'Hi there!'

>>> ""

" >>> 'R'

'R'

Note that the shell prints a string using single quotes, even when you enter it using double quotes. In this book, we use single quotes with single-character strings and double quotes with the empty string or with multi-character strings.

When working with strings, the programmer sometimes must be aware of a string's length and the positions of the individual characters within the string. A string's length is the number of characters it contains.

Python's len function returns this value when it is passed a string, as shown in the following session:

```
>>> len("Hi there!")
9
>>> len("")
0
```

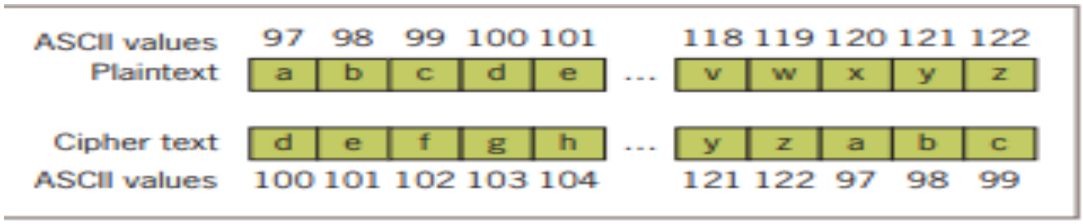
The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right.

Data Encryption

As you might imagine, data traveling on the Internet is vulnerable to spies and potential thieves. It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions.

For example, a person can sit in a car in the parking lot outside any major hotel and pick up transmissions between almost any two computers if that person runs the right sniffing software. For this reason, most applications now use data encryption to protect information transmitted on networks. Some application protocols include secure versions that use data encryption.

Examples of such versions are FTPS and HTTPS, which are secure versions of FTP and HTTP for file transfer and Web page transfer, respectively. Encryption techniques are as old as the practice of sending and receiving messages. The sender encrypts a message by translating it to a secret code, called a cipher text. At the other end, the receiver decrypts the cipher text back to its original plaintext form. Both parties to this transaction must have at their disposal one or more keys that allow them to encrypt and decrypt messages.



Recall that the ord function returns the ordinal position of a character value in the ASCII sequence, whereas chr is the inverse function.

""" File: encrypt.py

Encrypts an input string of lowercase letters and prints

the result. The other input is the distance value. """

plainText = input("Enter a one-word, lowercase message: ")

distance = int(input("Enter the distance value: "))

code = ""

for ch in plainText:

ordvalue = ord(ch)

cipherValue = ordvalue + distance

if cipherValue > ord('z'):

cipherValue = ord('a') + distance - \ (ord('z') - ordvalue + 1)

code += chr(cipherValue)

print(code)

"""

File: decrypt.py Decrypts an input string of lowercase letters and prints the result.

The other input is the distance value. """

```
code = input("Enter the coded text: ")

 distance = int(input("Enter the distance value: "))

 plainText = "" for ch in code: ordvalue = ord(ch) cipherValue = ordvalue - distance if cipherValue

 < ord('a'): cipherValue = ord('z') - \ (distance - (ord('a') - ordvalue - 1))

 plainText += chr(cipherValue)

 print(plainText)
```

Here are some executions of the two scripts in the IDLE shell:

Enter a one-word, lowercase message:

invaders Enter the distance value: 3

Lqydghuv Enter the coded text: lqydghuv

Enter the distance value: 3 invaders

These scripts could easily be extended to cover all of the characters, including spaces and punctuation marks

Strings and Number Systems

When you perform arithmetic operations, you use the decimal number system. This system, also called the base ten number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits.

As we saw in Chapter 1, the binary number system is used to represent all information in a digital computer. The two digits in this base two number system are 0 and 1.

Because binary numbers can be long strings of 0s and 1s, computer scientists often use other number systems, such as octal (base eight) and hexadecimal (base 16) as shorthand for these numbers.

To identify the system being used, you attach the base as a subscript to the number.

For example, the following numbers represent the quantity 41510 in the binary, octal, decimal, and hexadecimal systems

415 in binary notation 1100111112

415 in octal notation 6378

415 in decimal notation 41510

415 in hexadecimal notation $19F16$

The digits used in each system are counted from 0 to n – 1, where n is the system's base. Thus, the digits 8 and 9 do not appear in the octal system.

To represent digits with values larger than 910, systems such as base 16 use letters. Thus, A16 represents the quantity 1010, whereas 1016 represents the quantity 1610. In this section, we examine how these systems represent numeric quantities and how to translate from one notation to another.

String Methods

Text processing involves many different operations on strings. For example, consider the problem of analyzing someone's writing style. Short sentences containing short words are generally considered more readable than long sentences containing long words. A program to compute a text's average sentence length and the average word length might provide a rough analysis of style. Let's start with counting the words in a single sentence and finding the average word length. This task requires locating the words in a string. Fortunately, Python includes a set of string operations called methods that make tasks like this one easy. In the next session, we use the string method split to obtain a list of the words contained in an input string. We then print the length of the list, which equals the number of words, and compute and print the average of the lengths of the words in the list.

```
>>> sentence = input("Enter a sentence: ")
```

Enter a sentence: This sentence has no long words.

>>> listOfWords = sentence.split()

>>> print("There are", len(listOfWords), "words.")

There are 6 words.

>>> sum = 0 >>> for word in listOfWords

sum += len(word)

>>> print("The average word length is", sum / len(listOfWords))

<an object>.<method name>.(<argument-1><argument-n>)

Methods can also expect arguments and return values. A method knows about the internal state of the object with which it is called.

Thus, the method split in our example builds a list of the words in the string object to which sentence refers and returns it. In short, methods are as useful as functions, but you need to get used to the dot notation, which you have already seen when using a function associated with a module.

In Python, all data values are in fact objects, and every data type includes a set of methods to use with objects of that type

Reading and Writing to text files in Python

Python provides inbuilt functions for creating, writing, and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s, and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character (‘\n’) in python by default.
- **Binary files:** In this type of file, there is no terminator for a line, and the data is stored after converting it into machine-understandable binary language.

In this article, we will be focusing on opening, closing, reading, and writing data in a text file.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.
2. **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
3. **Write Only ('w') :** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read ('w+')** : Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only ('a')**: Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

How Files are Loaded into Primary Memory

There are two kinds of memory in a computer i.e. Primary and Secondary memory every file that you saved or anyone saved is on secondary memory cause any data in primary memory is deleted when the computer is powered off. So when you need to change any text file or just to work with them in python you need to load that file into primary memory. Python interacts with files loaded in primary memory or main memory through **"file handlers"** (This is how your operating system gives access to python to interact with the file you opened by searching the file in its memory if found it returns a file handler and then you can work with the file).

Opening a File

It is done using the open() function. No module is required to be imported for this function.

File_object = open(r"File_Name","Access_Mode")

The file should exist in the same directory as the python program file else, the full address of the file should be written in place of the filename. Note: The **r** is placed before the filename to prevent the characters in the filename string to be treated as special characters. For example, if there is \temp in the file address, then \t is treated as the tab character, and an error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in the same directory and the address is not being placed.

- Python

```
# Open function to open the file "MyFile1.txt"

# (same directory) in append mode and

file1 = open("MyFile1.txt","a")



# store its reference in the variable file1

# and "MyFile2.txt" in D:\Text in file2

file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as an object for MyFile1 and file2 as object for MyFile2

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

File_object.close()

- Python

```
# Opening and Closing a file "MyFile.txt"

# for object name file1.

file1 = open("MyFile.txt","a")

file1.close()
```

Writing to a file

There are two ways to write in a file.

1. **write() :** Inserts the string str1 in a single line in the text file.

File_object.write(str1)

1. **writelines() :** For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File_object.read([n])

1. **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

File_object.readline([n])

1. **readlines() :** Reads all the lines and return them as each line a string element in a list.

File_object.readlines()

Note: '\n' is treated as a special character of two bytes

- Python3

```
# Program to show various ways to read and

# write data in a file.

file1 = open("myfile.txt","w")

L = ["This is Delhi \n","This is Paris \n","This is London \n"]



# \n is placed to indicate EOL (End of Line)

file1.write("Hello \n")

file1.writelines(L)

file1.close() #to change file access modes



file1 = open("myfile.txt","r+")



print("Output of Read function is ")

print(file1.read())

print()
```

```
# seek(n) takes the file handle to the nth

# byte from the beginning.

file1.seek(0)



print( "Output of Readline function is ")

print(file1.readline())

print()



file1.seek(0)



# To show difference between read and readline

print("Output of Read(9) function is ")

print(file1.read(9))

print()



file1.seek(0)
```

```
print("Output of Readline(9) function is ")

print(file1.readline(9))



file1.seek(0)

# readlines function

print("Output of Readlines function is ")

print(file1.readlines())

print()

file1.close()
```

Output:

Output of Read function is
Hello
This is Delhi
This is Paris
This is London

Output of Readline function is
Hello

Output of Read(9) function is
Hello

Th

Output of Readline(9) function is
Hello

Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']

Appending to a file

- Python3

```
# Python program to illustrate

# Append vs write mode

file1 = open("myfile.txt","w")

L = ["This is Delhi \n","This is Paris \n","This is London \n"]

file1.writelines(L)

file1.close()



# Append-adds at last

file1 = open("myfile.txt","a")#append mode

file1.write("Today \n")

file1.close()
```

```
file1 = open("myfile.txt","r")

print("Output of Readlines after appending")

print(file1.readlines())

print()

file1.close()



# Write-Overwrites

file1 = open("myfile.txt","w")#write mode

file1.write("Tomorrow \n")

file1.close()



file1 = open("myfile.txt","r")

print("Output of Readlines after writing")

print(file1.readlines())

print()

file1.close()
```

Output:

Output of Readlines after appending

['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing

['Tomorrow \n']

Python Lists

```
mylist = ["apple", "banana", "cherry"]
```

List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

ExampleGet your own Python Server

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

List Length

To determine how many items a list has, use the len() function:

Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

List Items - Data Types

List items can be of any data type:

Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

type()

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```

Example

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

The list() Constructor

It is also possible to use the list() constructor when creating a new list.

Example

Using the list() constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

- When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python Dictionaries

```
thisdict =	{
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

ExampleGet your own Python Server

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Ordered or Unordered?

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Python Functions

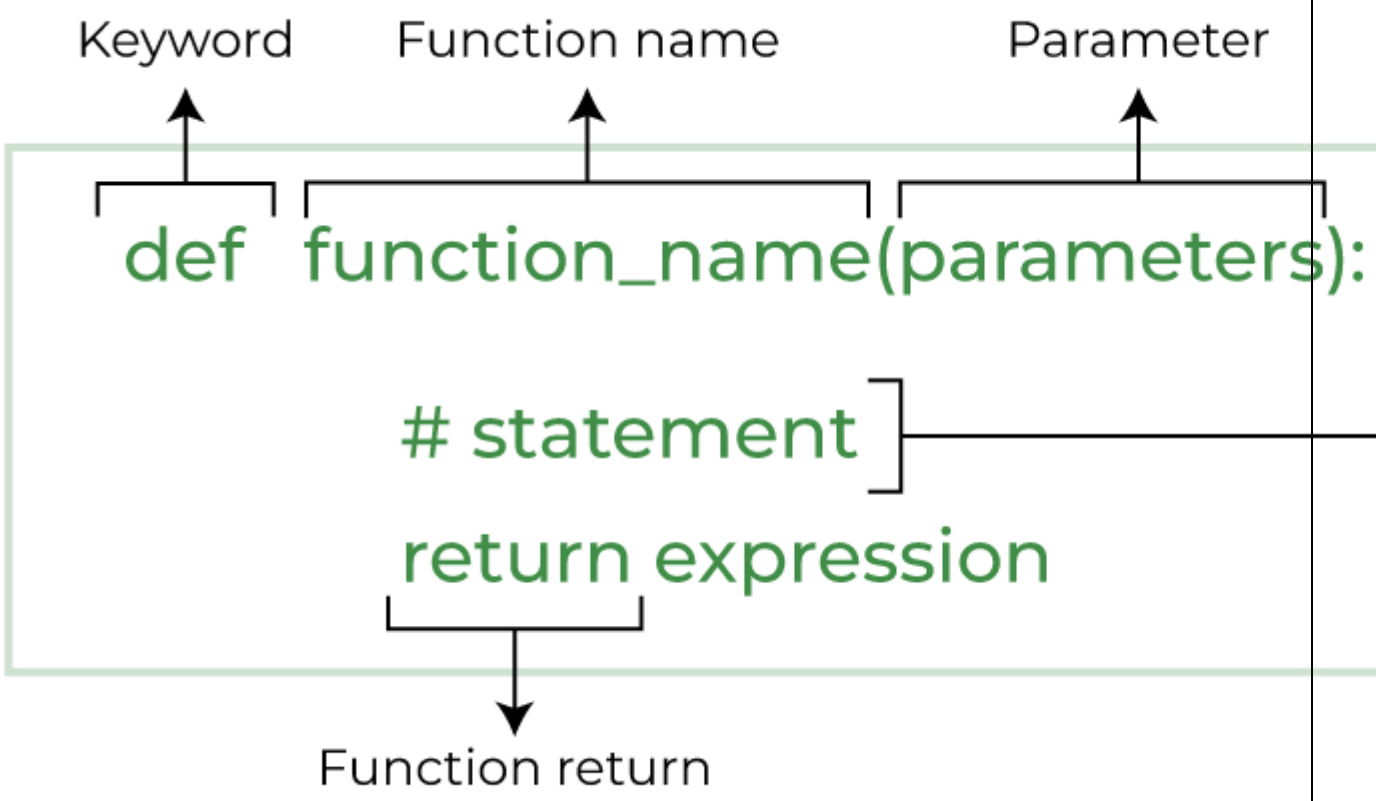
Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some **Benefits of Using Functions**

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Syntax of Python Function Declaration

Types of Functions in Python

There are mainly two types of functions in Python.

- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

Creating a Function in Python

We can create a user-defined function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require.

- Python3

```
# A simple Python function


def fun():
  print("Welcome to GFG")
```

Calling a Python Function

After creating a function in Python we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

- Python3

```
# A simple Python function

def fun():
    print("Welcome to GFG")



# Driver code to call a function

fun()
```

Output:

Welcome to GFG

Python Function with Parameters

If you have experience in C/C++ or Java then you must be thinking about the *return type* of the function and *data type* of arguments. That is possible in Python as well (specifically for Python 3.5 and above).

Defining and calling a function with parameters

```
def function_name(parameter: data_type) -> return_type:
    """Docstring"""
    # body of the function
    return expression
```

The following example uses arguments and parameters that you will learn later in this article so you can come back to it again if not understood.

- Python3

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2

    return num3


# Driver code
num1, num2 = 5, 15
```

```
ans = add(num1, num2)

print(f"The addition of {num1} and {num2} results {ans}.")
```

Output:

The addition of 5 and 15 results 20.

Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

In this example, we will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.

- Python3

```
# A simple Python function to check

# whether x is even or odd

def evenOdd(x):

    if (x % 2 == 0):

        print("even")

    else:

        print("odd")
```

```
# Driver code to call the function

evenOdd(2)

evenOdd(3)
```

Output:

even

odd

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- **Default argument**
- **Keyword arguments (named arguments)**
- **Positional arguments**
- **Arbitrary arguments** (variable-length arguments *args and **kwargs)

Let's discuss each type in detail.

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

- Python3

```
# Python program to demonstrate

# default arguments

def myFun(x, y=50):
```

```
    print("x: ", x)

    print("y: ", y)




# Driver code (We call myFun() with only

# argument)

myFun(10)
```

Output:

x: 10

y: 50

Like C++ default arguments, any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

- Python3

```
# Python program to demonstrate Keyword Arguments

def student(firstname, lastname):
```

```
    print(firstname, lastname)




# Keyword arguments

student(firstname='Geeks', lastname='Practice')

student(lastname='Practice', firstname='Geeks')
```

Output:

Geeks Practice

Geeks Practice

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

- Python3

```
def nameAge(name, age):

    print("Hi, I am", name)

    print("My age is ", age)
```

```
# You will get correct output because

# argument is given in order

print("Case-1:")

nameAge("Suraj", 27)

# You will get incorrect output because

# argument is not in order

print("\nCase-2:")

nameAge(27, "Suraj")
```

Output:

Case-1:

Hi, I am Suraj

My age is 27

Case-2:

Hi, I am 27

My age is Suraj

Arbitrary Keyword Arguments

In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)

Example 1: Variable length non-keywords argument

- Python3

```
# Python program to illustrate

# *args for variable number of arguments

def myFun(*argv):

    for arg in argv:

        print(arg)




myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

Hello

Welcome

to

GeeksforGeeks

Example 2: Variable length keyword arguments

- Python3

```
# Python program to illustrate

# *kwargs for variable number of keyword arguments




def myFun(**kwargs):

    for key, value in kwargs.items():

        print("%s == %s" % (key, value))




# Driver code

myFun(first='Geeks', mid='for', last='Geeks')
```

Output:

first == Geeks

mid == for

last == Geeks

Docstring

The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function:

Syntax: print(function_name.__doc__)

Example: Adding Docstring to the function

- Python3

```
# A simple Python function to check

# whether x is even or odd




def evenOdd(x):

    """Function to check if the number is even or odd"""



    if (x % 2 == 0):

        print("even")

    else:

        print("odd")




# Driver code to call the function
```

```
print(evenOdd.__doc__)
```

Output:

Function to check if the number is even or odd

Python Function within Functions

A function that is defined inside another function is known as the inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

- Python3

```
# Python program to

# demonstrate accessing of

# variables of nested functions



def f1():

    s = 'I love GeeksforGeeks'



    def f2():

        print(s)
```

```
    f2()



# Driver's code

f1()
```

Output:

I love GeeksforGeeks

Anonymous Functions in Python

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

- Python3

```
# Python code to illustrate the cube of a number

# using lambda function

def cube(x): return x*x*x



cube_v2 = lambda x : x*x*x



print(cube(7))

print(cube_v2(7))
```

Output:

343

343

Return Statement in Python Function

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller. **The syntax for the return statement is:**

return [expression_list]

The return statement can consist of a variable, an expression, or a constant which is returned at the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example: Python Function Return Statement

- Python3

```
def square_value(num):
    """This function returns the square
    value of the entered number"""
    return num**2



print(square_value(2))
print(square_value(-4))
```

Output:

4

16

A Quick Review of What Functions

Are and How They Wor

We have been using built-in functions since Chapter 2, and we very briefly discussed how to define functions in Chapter 5 so we could use them in some case studies. Before we delve into the use of functions in designing programs, it will be a good idea to review what you have learned about functions thus far.

1. A function packages an algorithm in a chunk of code that you can call by name. For example, the reply function in the doctor program of Chapter 5 builds and returns a doctor's reply to a patient's sentence.
2. 2. A function can be called from anywhere in a program's code, including code within other functions. During program execution, there may be a complex chain of function calls, where one function calls another and waits for its results to be returned, and so on. For example, in the doctor program, the main function calls the reply function, which in turn calls the changePerson function. The result of changePerson is returned to reply, whose result is returned to main.
3. 3. A function can receive data from its caller via arguments. For example, the doctor program's reply function expects one argument—a string representing the patient's sentence. However, some functions, like those of the sentence generator program of Chapter 5, need no arguments to do their work.
4. 4. When a function is called, any expressions supplied as arguments are first evaluated. Their values are copied to temporary storage locations named by the parameters in the function's definition. The parameters play the same role as variables in the code that the function then executes.
5. 5. A function may have one or more return statements, whose purpose is to terminate the execution of the function and return control to its caller. A return statement may be followed by an expression. In that case, Python evaluates the

expression and makes its value available to the caller when the function stops execution.

6. For example, the doctor program's reply function returns either the value returned by the random. choice function or the value returned by the change Person function. If a function does not include a return statement, Python automatically returns the value None to the caller. With these reminders about the use and behavior of functions under your belt, you are now ready to tackle the finer points of program design with functions.

Problem Solving with Top-Down Design

One popular design strategy for programs of any significant size and complexity is called top-down design. This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as problem decomposition.

As each subproblem is isolated, its solution is assigned to a function. Problem decomposition may continue down to lower levels, because a subproblem might in turn contain two or more lower-level problems to solve.

As functions are developed to solve each subproblem, the solution to the overall problem is gradually filled out in detail. This process is also called stepwise refinement.

The Design of the Text-Analysis Program

The program requires simple input and output components, so these can be expressed as statements within a main function.

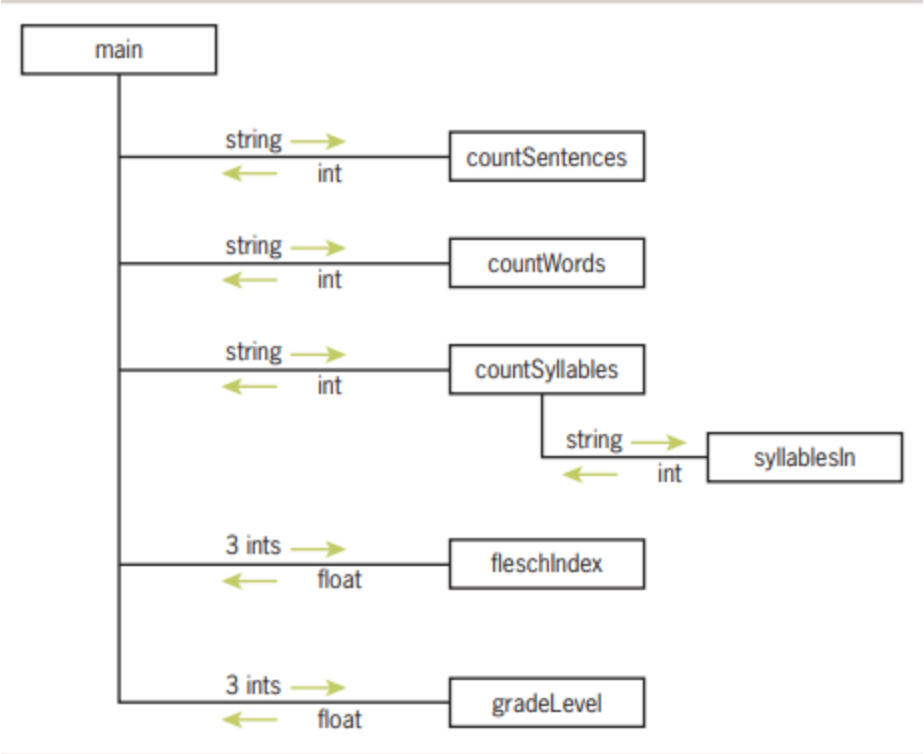
However, the processing of the input is complex enough to decompose into smaller sub processes, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores.

Generally, you develop a new function for each of these computational tasks. The relationships among the functions in this design are expressed in the structure chart shown. Each box in the structure chart is labeled with a function name

The main function at the top is where the design begins, and decomposition leads us to the lower-level functions on which main depends.

The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them.

For example, the function count Sentences takes a string as an argument and returns the number of sentences in that string.

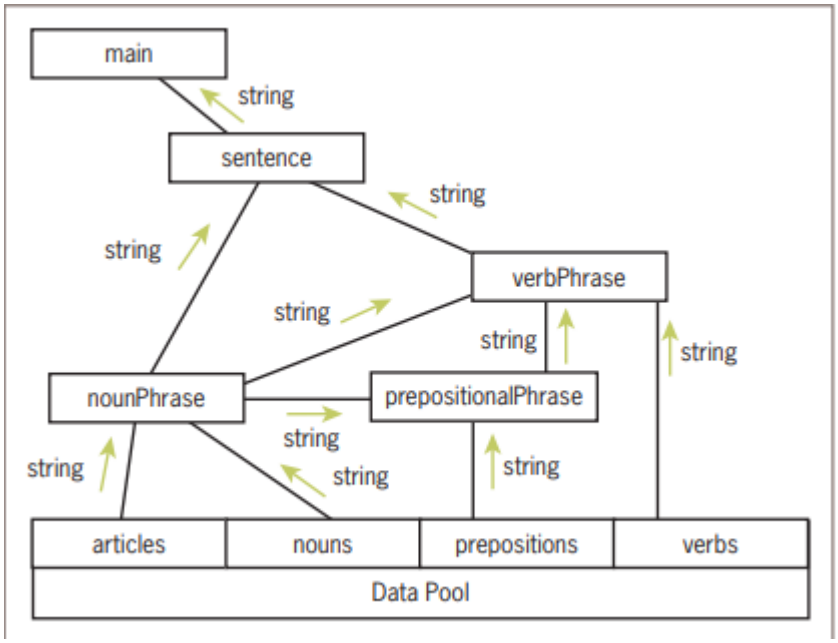


A structure chart for the text-analysis program

The Design of the Sentence-Generator Program

From a global perspective, the sentence-generator program (Section 5.3) consists of a main loop in which sentences are generated a user-specified number of times, until the user enters 0. The I/O and loop logic are simple enough to place in the main function. The rest of the design involves generating a sentence. Here, you decompose the problem by simply following the grammar rules for phrases. To generate a sentence, you generate a noun phrase followed by a verb phrase, and so on. Each of the grammar rules poses a problem that is solved by a single function.

The structure of a problem can often give you a pattern for designing the structure of the program to solve it. In the case of the sentence generator, the structure of the problem comes from the grammar rules, although they are not explicit data structures in the program. In later chapters, we will see many examples of program designs that also mirror the structure of the data being processed

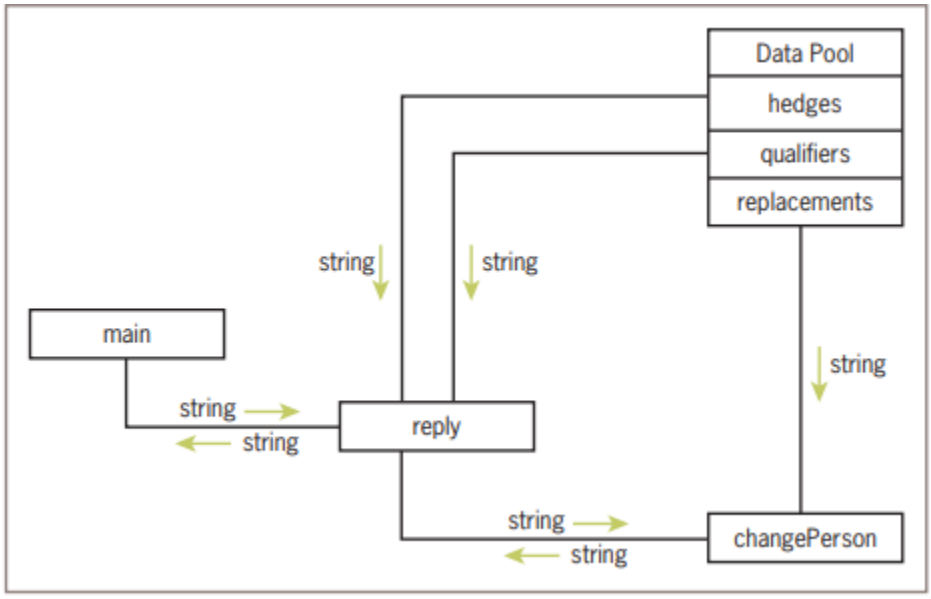


A structure chart for the sentence-generator program

The Design of the Doctor Program

At the top level, the designs of the doctor program (Section 5.5) and the sentence-generator program are similar. Both programs have main loops that take a single user input and print a result

The doctor program processes the input by responding to it as an agent would in a conversation. Thus, the responsibility for responding is delegated to the reply function. Note that the two functions main and reply have distinct responsibilities.



A structure chart for the doctor program

Design with Recursive Functions

In top-down design, you decompose a complex problem into a set of simpler problems and solve these with different functions. In some cases, you can decompose a complex problem into smaller problems of the same form. In these cases, the subproblems can all be solved by using the same function. This design strategy is called recursive design, and the resulting functions are called recursive functions.

Recursion in Python

The term Recursion can be defined as the process of defining something in terms of itself.

In simple words, it is a process in which a function calls itself directly or indirectly.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Syntax:

```
def func(): <--
            |
            | (recursive call)
            |
    func() ----
```

Example 1: A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8….

Python3

```
# Program to print the fibonacci series upto n_terms



# Recursive function

def recursive_fibonacci(n):

  if n <= 1:
```

```
    return n
  else:
    return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))


n_terms = 10


# check if the number of terms is valid
if n_terms <= 0:
  print("Invalid input ! Please input a positive value")
else:
  print("Fibonacci series:")
for i in range(n_terms):
   print(recursive_fibonacci(i))
```

Output

Fibonacci series:

0

1

1

2

3

5

8

13

21

34

Example 2: The factorial of 6 is denoted as 6! = 1*2*3*4*5*6 = 720.

Python3

```
# Program to print factorial of a number

# recursively.



# Recursive function

def recursive_factorial(n):

  if n == 1:

      return n

  else:

      return n * recursive_factorial(n-1)



# user input
```

```
num = 6



# check if the input is valid or not

if num < 0:

  print("Invalid input ! Please enter a positive number.")

elif num == 0:

  print("Factorial of number 0 is 1")

else:

  print("Factorial of number", num, "=", recursive_factorial(num))
```

Output

Factorial of number 6 = 720

What is Tail-Recursion?

A unique type of recursion where the last procedure of a function is a recursive call. The recursion may be automated away by performing the request in the current stack frame and returning the output instead of generating a new stack frame. The tail-recursion may be optimized by the compiler which makes it better than non-tail recursive functions.

Is it possible to optimize a program by making use of a tail-recursive function instead of non-tail recursive function?

Considering the function given below in order to calculate the factorial of n, we can observe that the function looks like a tail-recursive at first but it is a non-tail-recursive function. If we observe closely, we can see that the value returned by Recur_facto(n-1) is used in Recur_facto(n), so the call to Recur_facto(n-1) is not the last thing done by Recur_facto(n).

Python3

```
# Program to calculate factorial of a number

# using a Non-Tail-Recursive function.



# non-tail recursive function

def Recur_facto(n):



    if (n == 0):

        return 1



    return n * Recur_facto(n-1)



# print the result

print(Recur_facto(6))
```

Output

720

We can write the given function Recur_facto as a tail-recursive function. The idea is to use one more argument and in the second argument, we accommodate the value of the factorial. When n reaches 0, return the final value of the factorial of the desired number.

Python3

```
# Program to calculate factorial of a number

# using a Tail-Recursive function.



# A tail recursive function

def Recur_facto(n, a = 1):



    if (n == 0):

        return a



    return Recur_facto(n - 1, n * a)



# print the result

print(Recur_facto(6))
```

Output

720

Namespaces and Scope in Python

What is namespace:

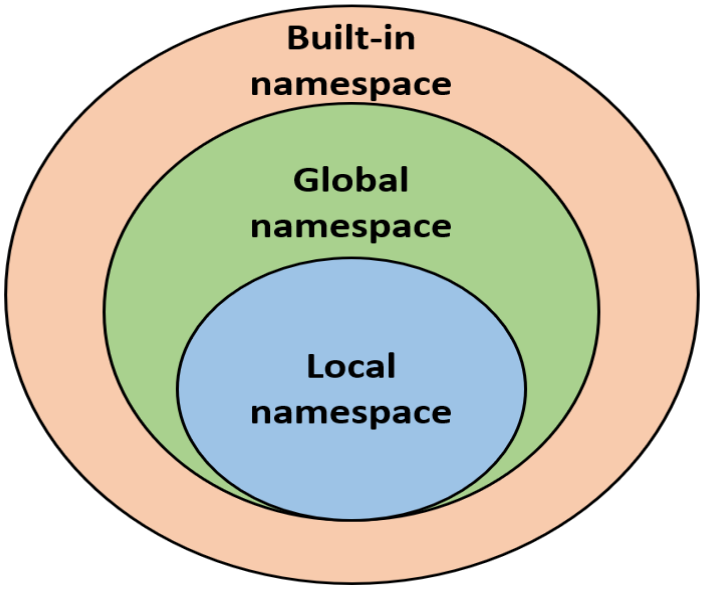
A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary. Let's go through an example, a directory-file system structure in computers. Needless to say, that one can have multiple directories having a file with the same name inside every directory. But one can get directed to the file, one wishes, just by specifying the absolute path to the file.

Real-time example, the role of a namespace is like a surname. One might not find a single "Alice" in the class there might be multiple "Alice" but when you particularly ask for "Alice Lee" or "Alice Clark" (with a surname), there will be only one (time being don't think of both first name and surname are same for multiple students).

On similar lines, the Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives a little more information. Its **Name** (which means name, a unique identifier) + **Space**(which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.

Types of namespaces :

When Python interpreter runs solely without any user-defined modules, methods, classes, etc. Some functions like print(), id() are always present, these are built-in namespaces. When a user creates a module, a global namespace gets created, later the creation of local functions creates the local namespace. The **built-in namespace** encompasses the **global namespace** and the global namespace encompasses the **local namespace**.



Type of Namespaces

The lifetime of a namespace :

A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace comes to an end. Hence, it is not possible to access the inner namespace's objects from an outer namespace.

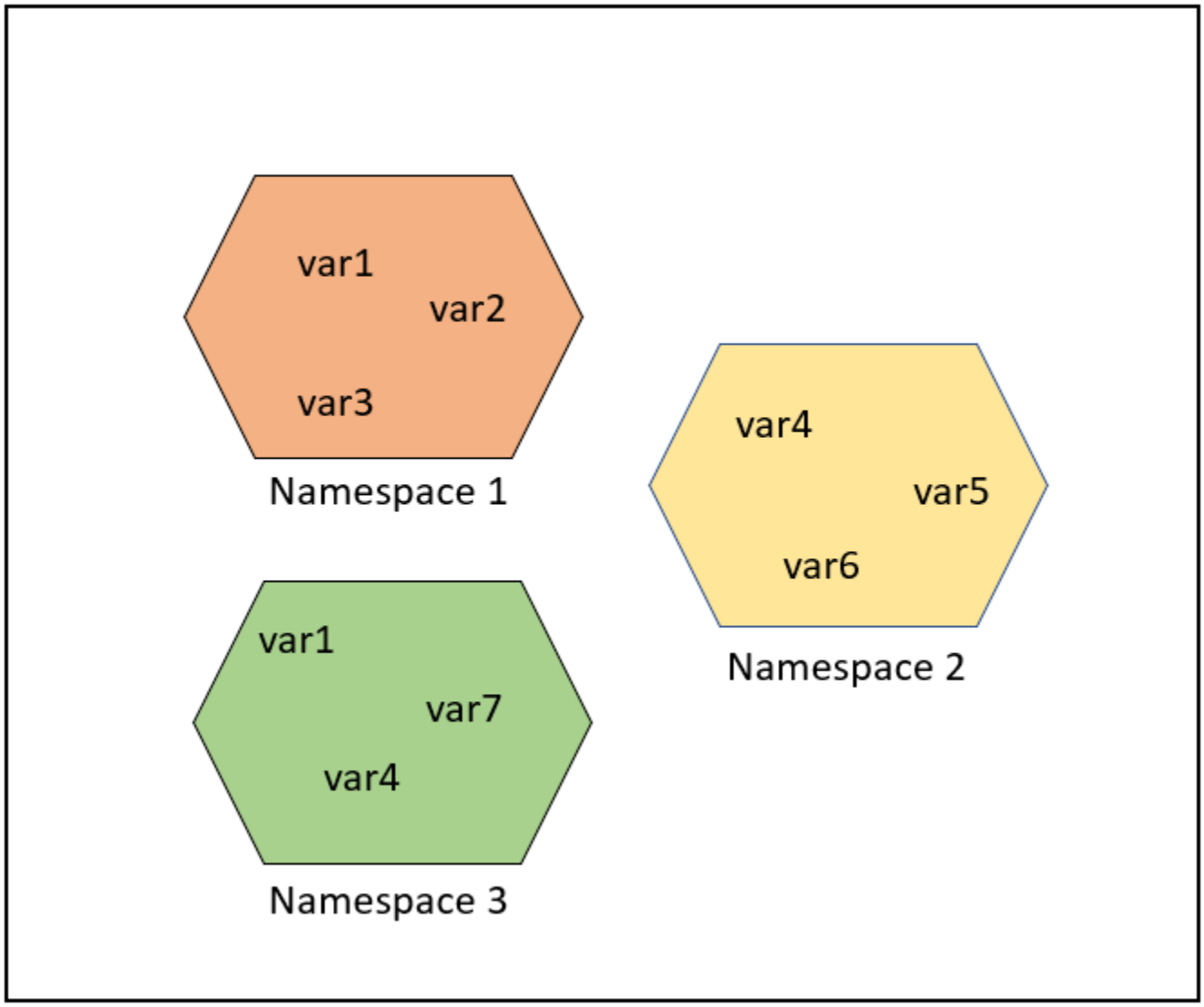
Example:

- Python3

```
# var1 is in the global namespace

var1 = 5

def some_func():



    # var2 is in the local namespace
```

```
var2 = 6

def some_inner_func():



    # var3 is in the nested local

    # namespace

    var3 = 7
```

As shown in the following figure, the same object name can be present in multiple namespaces as isolation between the same name is maintained by their namespace.



But in some cases, one might be interested in updating or processing global variables only, as shown in the following example, one should mark it explicitly as global and the update or process. Note that the line “count = count +1” references the global variable and therefore uses the global variable, but compare this to the same line written “count = 1”. Then the line “global count” is absolutely needed according to scope rules.

- Python3

```
# Python program processing

# global variable



count = 5

def some_method():

    global count

    count = count + 1

    print(count)

some_method()
```

Output:

6

Scope of Objects in Python :

Scope refers to the coding region from which a particular Python object is accessible. Hence one cannot access any particular object from anywhere from the code, the accessing has to be allowed by the scope of the object.

Let’s take an example to have a detailed understanding of the same:

Example 1:

- Python3

```
# Python program showing

# a scope of object



def some_func():

    print("Inside some_func")

    def some_inner_func():

        var = 10

        print("Inside inner function, value of var:",var)

    some_inner_func()

    print("Try printing var from outer function: ",var)

some_func()
```

Output:

```
Inside some_func

Inside inner function, value of var: 10


Traceback (most recent call last):

  File "/home/1eb47bb3eac2fa36d6bfe5d349dfcb84.py", line 8, in

    some_func()

  File "/home/1eb47bb3eac2fa36d6bfe5d349dfcb84.py", line 7, in some_func
```

```
    print("Try printing var from outer function: ",var)
```

NameError: name 'var' is not defined

Higher Order Functions in Python

A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions. It is worth knowing that this higher order function is applicable for functions and methods as well that takes functions as a parameter or returns a function as a result. Python too supports the concepts of higher order functions.

Properties of higher-order functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Functions as objects

In Python, a function can be assigned to a variable. This assignment does not call the function, instead a reference to that function is created. Consider the below example, for better understanding.

Example:

```
# Python program to illustrate functions

# can be treated as objects

def shout(text):

    return text.upper()
```

```
print(shout('Hello'))



# Assigning function to a variable

yell = shout



print(yell('Hello'))
```

Output:

HELLO

HELLO

In the above example, a function object referenced by shout and creates a second name pointing to it, yell.

Passing Function as an argument to other function

Functions are like objects in Python, therefore, they can be passed as argument to other functions. Consider the below example, where we have created a function greet which takes a function as an argument.

Example:

```
# Python program to illustrate functions

# can be passed as arguments to other functions

def shout(text):
```

```
    return text.upper()



def whisper(text):

    return text.lower()



def greet(func):

    # storing the function in a variable

    greeting = func("Hi, I am created by a function \

    passed as an argument.")

    print(greeting)



greet(shout)

greet(whisper)
```

Output:

HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.

hi, i am created by a function passed as an argument.

Returning function

As functions are objects, we can also return a function from another function. In the below example, the create_adder function returns adder function.

Example:

```
# Python program to illustrate functions

# Functions can return another function



def create_adder(x):

    def adder(y):

        return x + y



    return adder



add_15 = create_adder(15)



print(add_15(10))
```

Output:

25

Decorators

Decorators are the most common use of higher-order functions in Python. It allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax:

@gfg_decorator

def hello_decorator():

.

.

.

The above code is equivalent to –

def hello_decorator():

.

.

.

hello_decorator = gfg_decorator(hello_decorator)

In the above code, gfg_decorator is a callable function, will add some code on the top of some another callable function, hello_decorator function and return the wrapper function.

Example:

```
# defining a decorator

def hello_decorator(func):



    # inner1 is a Wrapper function in
```

```
    # which the argument is called



    # inner function can access the outer local

    # functions like in this case "func"

    def inner1():

        print("Hello, this is before function execution")



        # calling the actual function now

        # inside the wrapper function.

        func()



        print("This is after function execution")



    return inner1




# defining a function, to be called inside wrapper
```

```
def function_to_be_used():

    print("This is inside the function !!")




# passing 'function_to_be_used' inside the

# decorator to control its behavior

function_to_be_used = hello_decorator(function_to_be_used)




# calling the function

function_to_be_used()
```

Output:

Hello, this is before function execution

This is inside the function !!

This is after function execution